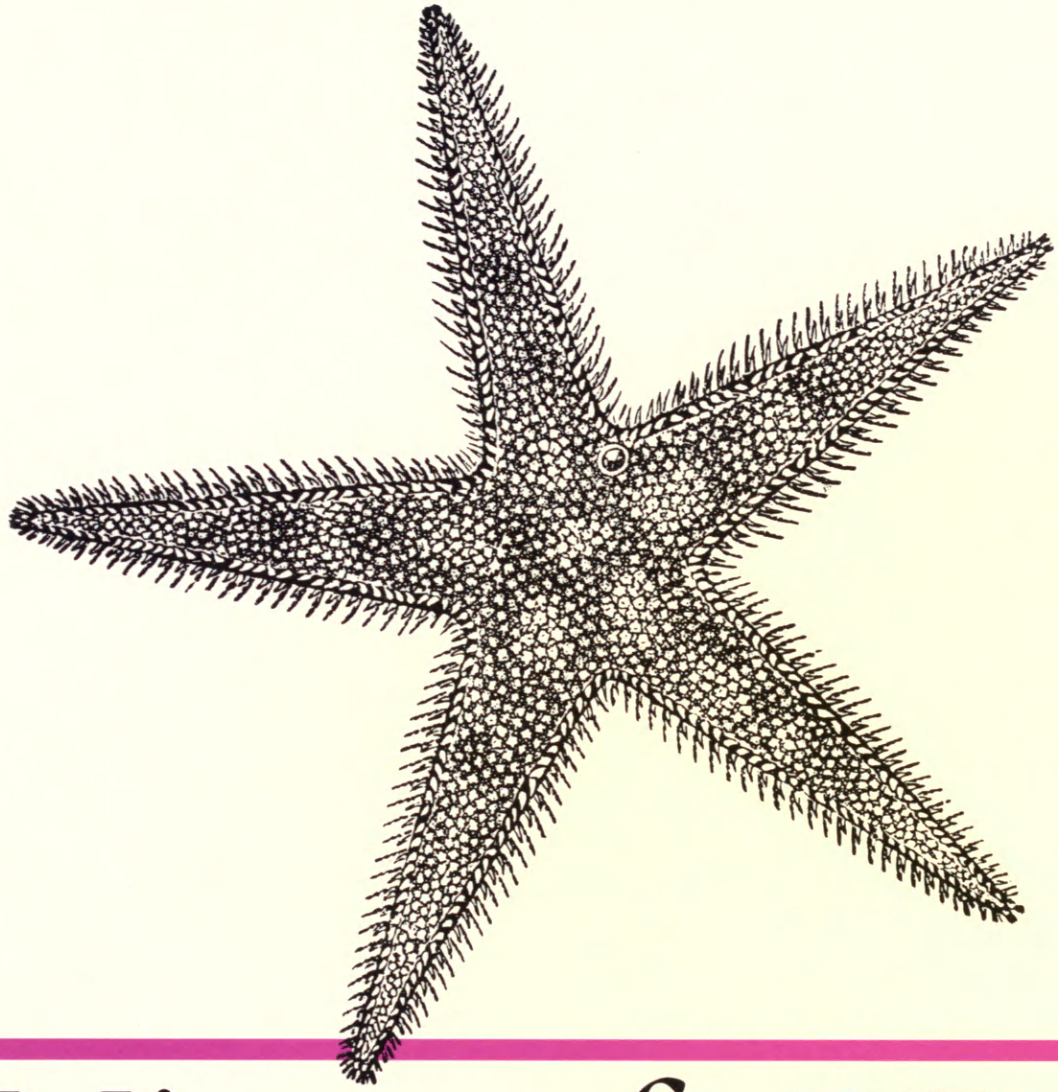
Distributed Applications



Microsoft RPC

Programming Guide

A NUTSHELL®

HANDBOOK

John Shirley & Ward Rosenberry

O'Reilly & Associates, Inc.

Microsoft RPC Programming Guide

Microsoft RPC Programming Guide

John Shirley and Ward Rosenberry
Digital Equipment Corporation

O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472

Microsoft RPC Programming Guide
by John Shirley and Ward Rosenberry

Copyright © 1995 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Editor: Andy Oram

Production Editor: Clairemarie Fisher O'Leary

Printing History:

March 1995: First Edition.

O'Reilly & Associates and the author specifically disclaim all warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for particular purpose with respect to the diskettes and the programs therein contained, and in no event shall O'Reilly & Associates or the author be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential, or other damages.

Nutshell Handbook and the Nutshell Handbook logo are registered trademarks of O'Reilly & Associates, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-070-8

Table of Contents

Preface ..... *xi*
Conventions xii
Book Organization xii
How to Use This Book xiii
Obtaining the Example Programs xiv
CompuServe xiv
FTP xvi
FTPMAIL xvii
BITFTP xvii
Acknowledgments xviii
Joint Venture xix

1: Overview of an RPC Application *1*

A Simple Interface 4
Universal Unique Identifiers 5
The Interface Definition 6
Stub and Header Generation Using the MIDL Compiler 7
A Simple Client 8
A Minimal Server 11
Remote Procedure Implementation 11
A Distributed Application Environment 13
Server Initialization 18
Producing the Application 21
Microsoft RPC Libraries 21

Compile and Link the Client and Server Code 21
Running the Application 23

2: Using a Microsoft RPC Interface 27

Microsoft Interface Definition Language (MIDL) 28
Attributes 28
Structure of an Interface Definition 29
Interface Header Attributes 29
The Inventory Application 30
Type Definitions, Data Attributes, and Constants 30
Procedure Declarations and Parameter Attributes 38
Using the MIDL Compiler 40
Generating Client Files 42
Generating Server Files 42
Using an ACF to Customize Interface Usage 42
Selecting a Binding Method 43
Controlling Errors and Exceptions 44
Excluding Unused Procedures 44

3: How to Write Clients 45

Binding 45
Implementing a Binding Method 46
Automatic Binding Management 49
Implicit Binding Management 50
Explicit Binding Management 52
Steps in Finding Servers 55
Finding a Protocol Sequence 56
Finding a Server Host 58
Finding an Endpoint 59
Interpreting Binding Information 60
Finding a Server from a Name Service Database 61
Finding a Server from Strings of Binding Data 64
Customizing a Binding Handle 66
Authentication 71
Error Parameters or Exceptions 72
Using Exception Handlers in Clients or Servers 72
Using Remote Procedure Parameters to Handle Errors 72
Compiling and Linking Clients 74
Local Testing 76

4: Pointers, Arrays, and Memory Usage *79*

Kinds of Pointers 79
Pointers as Output Parameters 80
Pointers as Input Parameters 82
Using Pointers to Pointers for New Output 84
Pointers as Procedure Return Values 86
Pointer Summary 87
Kinds of Arrays 90
Selecting a Portion of a Varying Array 90
Managing the Size of a Conformant Array 91
Memory Usage 94
Node-By-Node Allocation and Deallocation 96
Using Contiguous Server Memory 97
Allocating Buffers with the Client Application 97
Persistent Storage on the Server 98

5: How to Write a Server *99*

Some Background on Call Processing 99
Initializing the Server 101
Registering Interfaces 102
Creating Server Binding Information 104
Advertising the Server 107
Managing Server Endpoints 109
Listening for Remote Procedure Calls 110
Writing Remote Procedures 112
Managing Memory in Remote Procedures 113
Allocating Memory for Conformant Arrays 116
Compiling and Linking Servers 117

6: Using a Name Service *121*

Naming 122
DefaultEntry 122
Server Entries 123
Creating a Server Entry and Exporting Binding Information 125
Some Rules for Using the Microsoft Locator 126

7: Context Handles 129

The Remote_file Application 129
Declaring Context in an Interface Definition 130
Using a Context Handle in a Client 131
Binding Handles and Context Handles 133
Managing Context in a Server 133
Writing Procedures That Use a Context Handle 134
Writing a Context Rundown Procedure 136

A: MIDL and ACF Attributes Quick Reference 137

B: RPC Runtime Routines Quick Reference 143

C: The Arithmetic Application 149

How to Build and Run the Application 149
Application Files 150

D: The Inventory Application 157

How to Run the Application 158
Application Files 159

E: The Rfile Application 191

How to Run the Application 191
Application Files 192

F: The Windows Phonebook Application 201

How to Build and Run the Application 201
Application Files 201

Index 223

List of Figures

1–1 Client-server model 2
1–2 RPC mechanism 3
1–3 Application development 3
1–4 Arithmetic application: interface development 5
1–5 Arithmetic application: client development 9
1–6 Arithmetic application: server development 12
1–7 Binding 13
1–8 Binding information 13
1–9 Server initializing 15
1–10 Client finding a server 16
1–11 Completing a remote procedure call 17
1–12 Arithmetic application: complete development 25
2–1 Producing an interface 41
3–1 A comparison of binding management methods 47
3–2 How a customized binding handle works 67
3–3 Producing a client 75
5–1 How the server runtime library handles a call 100
5–2 Producing a server 118
6–1 Server entries in the name service database 123
6–2 A simple use of a name service database 125

Conventions

Throughout the book we use the following typographic conventions:

`Constant width`
: indicates a language construct such as a MIDL keyword, a code example, system output, or user input. Words in constant width also represent application-specific variables and procedures.

`Constant Bold`
: is used in examples to indicate text that is literally typed by the user.

Bold
: introduces new terms or concepts.

Italic
: in command syntax or examples indicates variables for which the user supplies a value. Italicized words in the text also represent system elements such as filenames and directory names, and user functions or RPC-specific routines.

[]
: enclose attributes in interface definitions and Attribute Configuration Files (ACFs) and are part of the syntax. Note that this is different from the common convention in which brackets enclose optional items in format and syntax descriptions.

`C:\>`
: represents system prompts.

`C:\SERVER>`
: represents a server system prompt to distinguish it from a client system prompt.

`C:\CLIENT>`
: represents a client system prompt to distinguish it from a server system prompt.

Book Organization

This book is divided into the following seven chapters and six appendices:

Chapter 1, *Overview of an RPC Application*, shows a complete, simple RPC application.

Chapter 2, *Using a Microsoft RPC Interface*, shows how to read an RPC interface definition (a file ending in *.idl*), which is a file that declares the remote procedures of an interface.

Chapter 3, *How to Write Clients*, discusses how to develop client programs for RPC interfaces. Topics include binding methods, finding servers, customizing binding handles, handling errors or exceptions, and compiling clients.

Chapter 4, *Pointers, Arrays, and Memory Usage*, shows how pointers and arrays are defined in an interface and how to develop applications to use them.

Chapter 5, *How to Write a Server*, discusses how to develop a server program for an RPC interface. Topics include initializing a server, writing remote procedures, and compiling servers.

Chapter 6, *Using a Name Service*, describes a name service database and how to use it with distributed applications.

Chapter 7, *Context Handles*, shows how to maintain a state (such as a file handle) on a specific server between remote procedure calls from a specific client.

Appendix A, *MIDL and ACF Attributes Quick Reference*, shows all the attributes in the Microsoft Interface Definition Language (MIDL) and Attribute Configuration File (ACF).

Appendix B, *RPC Runtime Routines Quick Reference*, shows all the RPC runtime routines organized into convenient categories.

Appendix C, *The Arithmetic Application*, is a small application that shows the basics of remote procedure calls.

Appendix D, *The Inventory Application*, is a somewhat richer application than that in Appendix C, showing different MIDL data types, how to use ACFs, and how to find servers by importing information from a name service database.

Appendix E, *The Rfile Application*, shows how to use context handles and how to find servers using strings of network location information.

Appendix F, *The Windows Phonebook Application*, offers a simple Windows-based client that uses RPC to get phone numbers from a database on the server.

How to Use This Book

If you are developing just a client for an existing RPC interface and server, read the following chapters first:

- Chapter 1, *Overview of an RPC Application*
- Chapter 2, *Using a Microsoft RPC Interface*
- Chapter 3, *How to Write Clients*

Read other chapters as needed to learn how to develop applications that use more features of interface definitions.

If you are developing a network interface with accompanying server, read the following:

- Chapter 1, *Overview of an RPC Application*
- Chapter 2, *Using a Microsoft RPC Interface*
- Chapter 3, *How to Write Clients*

```
Enter choice !off

Thank you for using CompuServe!

Off at 06:59 EST 11-Jan-95
Connect time = 0:06
```

FTP

To use FTP, you need a machine with direct access to the Internet. A sample session is shown, with what you should type in boldface.

```
% ftp ftp.uu.net
Connected to ftp.uu.net.
220 ftp.UU.NET FTP server (Version 6.34 Thu Oct 22 14:32:01 EDT 1992) ready.
Name (ftp.uu.net:andyo): anonymous
331 Guest login ok, send e-mail address as password.
Password: janetv@xyz.com (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/nutshell/ms_rpc
250 CWD command successful.
ftp> binary (Very important! You must specify binary transfer for compressed files.)
200 Type set to I.
ftp> prompt (Convenient, so you are not queried for every file transferred)
Interactive mode off.
ftp> mget *
200 PORT command successful.
        .
        .
        .
ftp> quit
221 Goodbye.
%
```

Each *.Z* archive contains all source code and configuration information required for building one example. Extract each example through a command like:

```
% zcat arith.dec94.tar.Z | tar xf -
```

System V systems require the following *tar* command instead:

```
% zcat arith.dec94.tar.Z | tar xof -
```

If *zcat* is not available on your system, use separate *uncompress* and *tar* commands.

The *tar* command creates a subdirectory that holds all the files from its archive. The *README.dec94* file in this subdirectory describes the goals of the example and how to build and run it; the text is an ASCII version of the introductory material from the corresponding appendix in this book.

FTPMAIL

FTPMAIL is a mail server available to anyone who can send electronic mail to and receive it from Internet sites. This includes any company or service provider that allows email connections to the Internet. Here's how you do it.

You send mail to *ftpmail@online.ora.com*. In the message body, give the FTP commands you want to run. The server will run anonymous FTP for you and mail the files back to you. To get a complete help file, send a message with no subject and the single word "help" in the body. The following is an example mail session that should get you the examples. This command sends you a listing of the files in the selected directory, and the requested example files. The listing is useful if there's a later version of the examples you're interested in.

```
% mail ftpmail@online.ora.com
Subject:
reply-to janetv@xyz.com                    Where you want files mailed
open
cd /published/oreilly/nutshell/ms_rpc
dir
get README.dec94
mode binary
uuencode                                   (or btoa if you have it)
get arith.dec94.tar.Z
get inv.dec94.tar.Z
get rfile.dec94.tar.Z
get phnbk.dec94.tar.Z
quit
.
```

A signature at the end of the message is acceptable as long as it appears after "quit."

All retrieved files will be split into 60KB chunks and mailed to you. You then remove the mail headers and concatenate them into one file, and then *uudecode* or *atob* it. Once you've got the desired *.Z* files, follow the directions under FTP to extract the files from the archive.

BITFTP

BITFTP is a mail server for BITNET users. You send it electronic mail messages requesting files, and it sends the files back to you by electronic mail. BITFTP currently serves only users who send it mail from nodes that are directly on BITNET, EARN, or NetNorth. BITFTP is a public service of Princeton University. Here's how it works:

To use BITFTP, send mail containing your *ftp* commands to *BITFTP@PUCC*. For a complete help file, send HELP as the message body.

The following is the message body you should send to BITFTP:

```
FTP  ftp.uu.net  NETDATA
USER  anonymous
PASS your Internet email address (not your bitnet address)
CD  /published/oreilly/nutshell/ms_rpc
DIR
GET README
BINARY
GET arith.dec94.tar.Z
GET inv.dec94.tar.Z
GET rfile.dec94.tar.Z
GET phnbk.dec94.tar.Z
QUIT
```

Once you've got the desired *.Z* files, follow the directions under FTP to extract the files from the archive. Since you are probably not on a UNIX system, you may need to get versions of *uudecode*, *uncompress*, *atob*, and *tar* for your system.

Questions about BITFTP can be directed to Melinda Varian, *MAINT@PUCC* on BITNET.

Acknowledgments

This book can be traced back to the DCE documentation set put out by Digital Equipment Corporation. John Shirley, working with Steve Talbott and Andy Oram from O'Reilly & Associates, wrote a DCE version of this book called *Guide to Writing DCE Applications*. Ward Rosenberry then took it over and thoroughly revised it to cover Microsoft RPC. While at first glance, it might seem that relatively little effort was required to write this new version, the work put into it was nevertheless considerable and required the cooperation and support of many individuals.

First off, I want to thank my editor at O'Reilly & Associates, Andy Oram, for his excellent advice and his persistence on this lengthy project.

For supporting this project I want to thank folks at Digital Equipment Corporation, in particular Jeff Shrieshiem, Frank Willison, and Michelle Chambers for funding various portions of the project. Also at Digital Equipment corporation, other major contributors to this book include Neil Miranda, who converted several DCE applications to Microsoft RPC Version 1.0 for use in this book. Riaz Zolfonoon later modified these applications for use with Microsoft RPC Version 2.0. Riaz also provided helpful advice on numerous aspects of Microsoft RPC.

Others at Digital who played central roles in developing the book include Jerry Harrow and Will Lees, who provided painstaking reviews of various drafts of sections of the book. Jim Teague provided a Microsoft RPC version of the phonebook application which was originally written for another O'Reilly book titled *Distributing Applications Across DCE and Windows NT*. Jim is a co-author of that book. Larry Friedman, Dick Annicchiarico, Michael Blackstock, Rob Philpott, and Andy Ferris provided bits and pieces of technical advice along the way. I also want to

thank Ladan Pooroshani, Beth Benoit, and Brian Shimpf for their cooperation and support.

Credit for logistical support goes to several folks at Digital including Gerry Fisher, Evelyn McKay, Lisa Cozins, and Madeline Cormier, all of whom made sure I had what I needed to get things done.

Several people at Microsoft Corporation also deserve thanks for providing various inputs to the book. These people include Debbie Black, Dave Tanaguchi, and Craig Link (from Microsoft's Win32 SDK forum on CompuServe).

Additional help and support for the DCE version of the book came from Tony Hinxman, Al Simons, David Magid, Margie Showman, Ken Ouellette, Mary Orcutt, Marll McDonald, Mark Heroux, Clem Cole, Marty Port, Ram Sudama, Diane Sherman, Susan Scott, David Strohmeyer, Karol McIntyre, Wei Hu, Susan Hunziker, Vicki Janicki, Beth Martin, Dan Cobb, Lois Frampton, Steve Miller, Eric Jendrock, Gary Schmitt, Ellen Vliet, Judy Davies, Judy Egan, Collis Jackson, David Kenney, Suzanne Lipsky, Darrell Icenogle, Terry Tvrdik, Howard Mayberry, and John Shirley's wife, Linda McClary.

Joe Scandora was very helpful on the Microsoft version of the book.

Book design and production credits go to lots of the folks at O'Reilly & Associates who artfully turned many pieces of a stark manuscript into a real book. Edie Freedman designed the cover. Jeff Robbins and Chris Reilley created the figures. Kismet McDonough, Eileen Kramer, and Clairemarie Fisher O'Leary did the copy-editing and production management. Kiersten Nauman assisted with the production work. Seth Maislin refined the index.

Finally, I want to thank Frank Willison for giving me the opportunity to work on this book.

Joint Venture

This book was produced as a cooperative effort between Digital Equipment Corporation and O'Reilly & Associates. While we at O'Reilly & Associates frequently work closely with vendors of hardware and software, this book gave us an opportunity for much more extensive cooperation and mutual support than is customary. It is a model we like, and we believe the end result testifies to the value of sharing one's resources in this way.

In this Chapter:
- *A Simple Interface*
- *A Simple Client*
- *A Minimal Server*
- *Producing the Application*
- *Running the Application*

1

Overview of an RPC Application

A traditional application is a single program running on a single computer system, where a procedure and its caller execute in the same address space. In contrast, the **client-server model** for distributed applications embodies a client program and a server program, usually running on different systems of a network. The client makes a request to the server, which is usually a continuously running daemon process, and the server sends a response back to the client (see Figure 1-1).

The **remote procedure call** (RPC) mechanism is the simplest way to implement client-server applications, because it keeps the details of network communications out of your application code. The idea is that each side behaves, as much as possible, the way it would within a traditional application: the programmer on the client side issues a call, and the programmer on the server side writes a procedure to carry out the desired function. To convey the illusion that you are working in a single address space, some hidden code has to handle all the networking. Many related issues are also involved, such as converting data between formats for different systems, and detecting communication errors.

Figure 1-2 shows the relationship between your application code and the RPC mechanism during a remote procedure call. In client application code, a remote procedure call looks like a local procedure call, because it is actually a call to a client stub. (A **stub** is surrogate code that supports remote procedure calls. Later in this chapter we'll discuss how stubs are created and what they do.) The **client stub** communicates with the **server stub** using the **RPC runtime library**, which is a set of standard runtime routines that supports all Microsoft RPC applications.

The server's RPC runtime library receives the remote procedure call and hands the client information to the server stub. The server stub invokes the remote procedure in the server application.

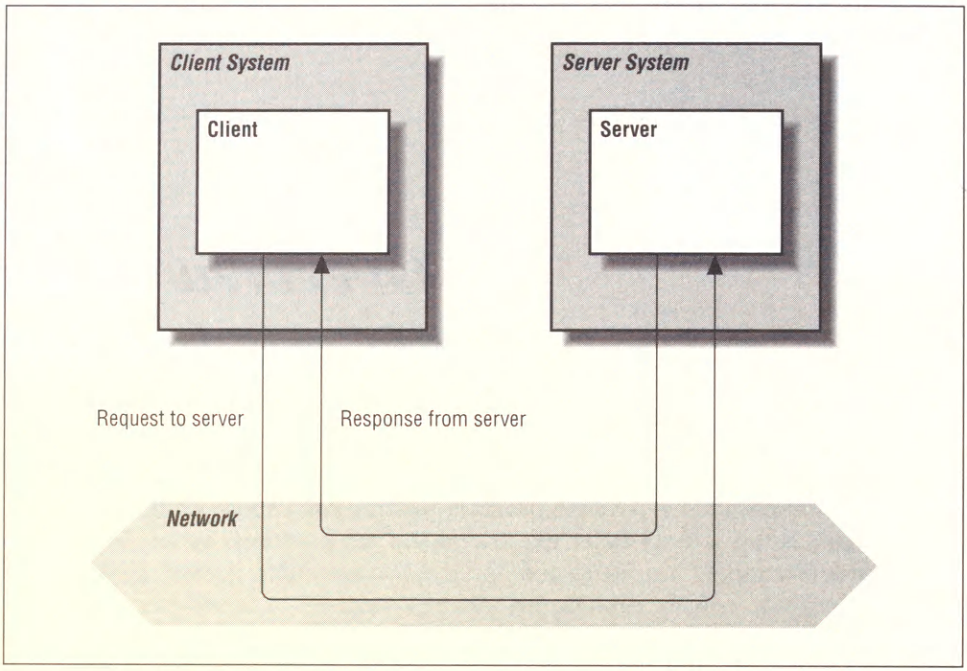


Figure 1–1. Client-server model

When the server finishes executing the remote procedure, its stub communicates output to the client stub, again by using the RPC runtime library. Finally, the client stub returns to the client application code.

Figure 1-3 shows the three phases required to develop a distributed application. An essential part of the RPC mechanism is an **interface**, which is a set of remote procedure declarations. Given the same interface, client and server development of an application can occur in parallel and on separate systems of the network.

In this chapter we will create an entire RPC application from scratch. Naturally, we'll use every shortcut and simplification the system offers to accomplish this feat. When you are done with the chapter, you will know the place of all the major RPC features, and how an application is developed.

You may not need to develop an entire application as shown in this chapter. If the interface and server already exist, your development may require only the client.

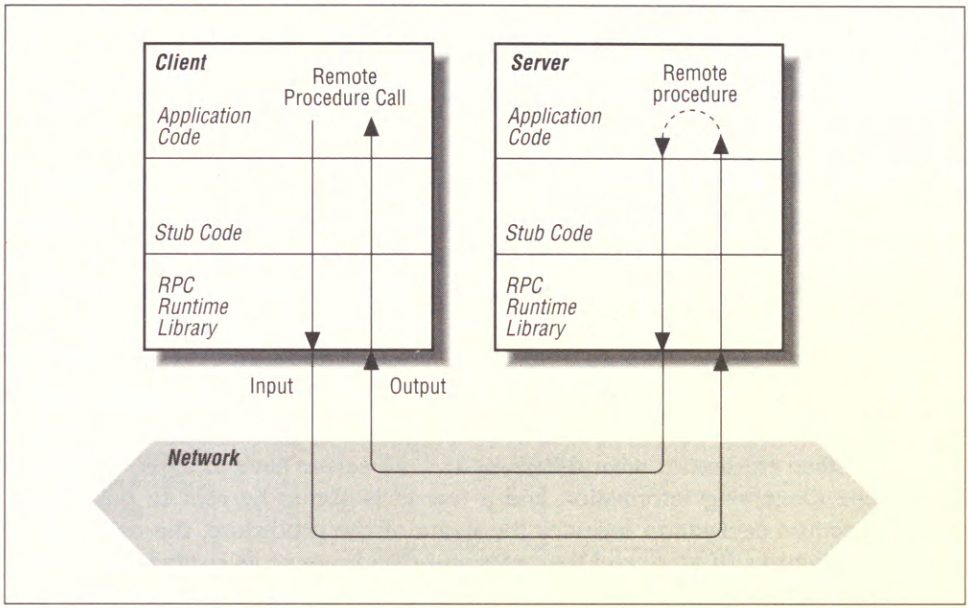


Figure 1–2. RPC mechanism

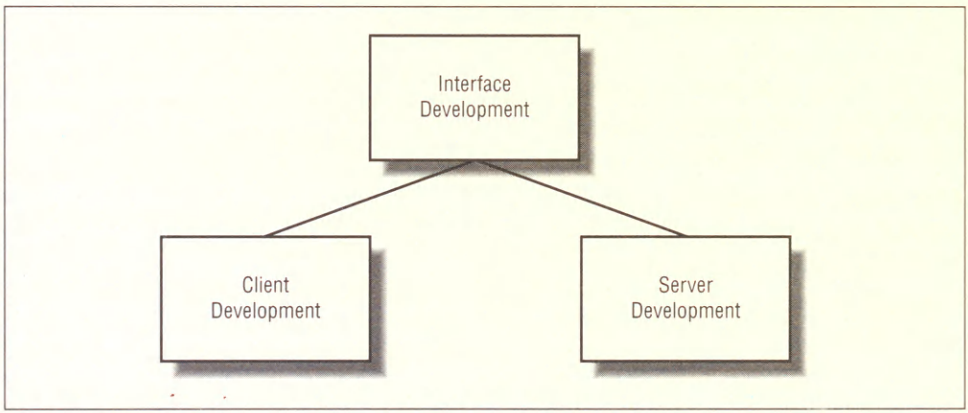


Figure 1–3. Application development

The arithmetic example in this chapter demonstrates a very simple one-client/one-server RPC application. Suppose a remote server system uses special hardware, such as an array processor. In our example, the client performs an arithmetic operation on arrays by calling a remote procedure that uses the array processor. The remote procedure executes on the server system, taking two arrays as arguments and adding together the elements of the arrays. The remote procedure returns the results to the client in a third array argument. Finally, the results of the remote procedure are displayed on the client system.

The arithmetic example is deliberately limited to demonstrate the basics of a distributed application implemented with RPC. We describe each portion of the application in this chapter, and Appendix C shows the complete code. The Preface tells you how to obtain source code online for this and other examples in the book.

A Simple Interface

When writing a local application, should you start by deciding exactly what functions you'll call and what arguments they take? Well, if you were dividing the work among multiple programmers and needed to clarify the interfaces between their work, you probably would proceed that way. The same reasoning applies to a distributed program: the client and server are being developed separately. Since the boundary or interface between them is the procedure call itself, you have to specify its attributes at the start.

So an interface consists of what the client and the server have to agree on; it contains some identifying information and a few facts about the remote procedures. Each **procedure declaration** includes the name of the procedure, the data type of the value it returns (if any), and the order and data types of its parameters (if any). An **interface definition** contains a set of procedure declarations and data types.

Just as programmers select functions from libraries, client application writers use interface definitions to determine how to call remote procedures. Server application writers use interface definitions to determine the data type of the remote procedure's return value, and the number, order, and data types of the arguments. The interface definition is like a design document that ties the client and server application code together. It is a formal definition describing the set of procedures offered by the interface.

You write the interface definition in **Microsoft Interface Definition Language** (MIDL). The MIDL closely resembles the declaration syntax and semantics of C, with the addition of attributes that allow information to be sent over a network.

You may think that we have introduced an unnecessary level of complexity here, but you will see that keeping the salient features of a distributed application in one file—the interface definition—makes it easier to scale up development to multiple servers and many clients for those servers.

Figure 1-4 shows the utilities used and the files produced when developing the arithmetic interface. The *uuidgen* utility generates a **universal unique identifier** (UUID) used in the interface definition to distinguish this interface from any other interface on the network. You use a text editor to write the rest of the interface definition, *arith.idl*. When the interface definition is complete, compile it with the MIDL compiler (*midl*) to generate stubs and a C header file that you use to develop the client and server programs.

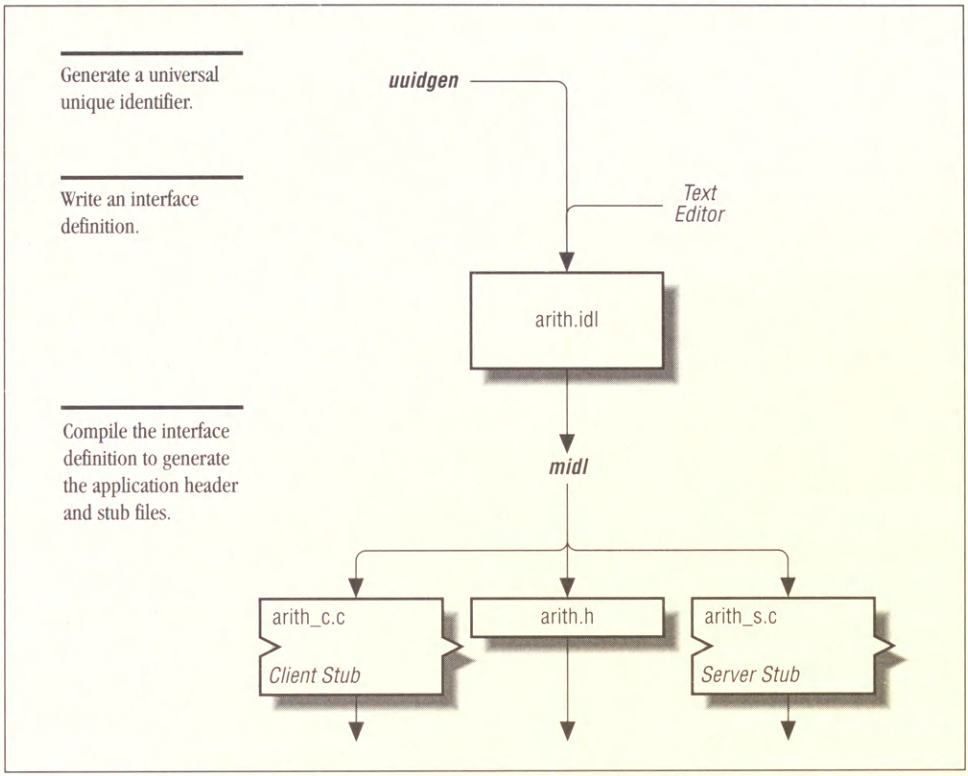


Figure 1–4. Arithmetic application: interface development

Universal Unique Identifiers

When you write a new interface, you must first generate a UUID with *uuidgen*. A UUID is simply a number that the *uuidgen* utility generates using time and network address information so that no matter when or where it is generated, it is guaranteed to be unique. A UUID is like a fingerprint that uniquely identifies something—such as an interface—across all network configurations.

An interface UUID is an excellent example of how you tie a client and server together through the MIDL file. When a client makes a remote procedure call, its UUID has to match that of the server. The RPC runtime library performs this check; this way you don't get unexpected results.

Generating a UUID in an interface definition template

To generate and display a UUID in a template for an interface definition, type the following command:

```
C:\> uuidgen -i
[
uuid(6AF85260-A3A4-101A-B1AE-08002B2E5B76),
version(1.0)
]
interface INTERFACENAME
{

}
```

In this example, the output appears at the terminal, but generally you save it in a file with the extension *.idl*. Replace the template name INTERFACENAME with a name you choose for the new interface. In the next section, we use a template like this to develop the arithmetic interface definition.

The Interface Definition

Now we are ready to write an interface definition. Here we put data type definitions and procedure declarations that need to be shared between server and client. Later, the MIDL compiler creates the header file and stubs from the interface definition, for use in your application.

The interface definition includes syntax elements called **attributes**, which specify features needed for distributed applications. Attributes convey information about the whole interface or items in the interface, including data types, arrays, pointers, structure members, union cases, procedures, and procedure parameters. For example, the in attribute specifies an input parameter for a remote procedure. You can pick out attributes in the file because they're enclosed in square brackets.

Example 1-1 shows a simple interface definition. The text consists of a **header** and **body**. The header contains a uuid attribute and the name assigned to the interface. The body specifies all procedures for the interface; it contains the procedure declarations with their data types and constants. There is only one procedure declared in our example. It adds two input arrays and returns the results in a third array.

Example 1-1: A Simple Interface Definition

```
/* FILE NAME: arith.idl */
/* This Interface Definition Language file represents a basic arithmetic */
/* procedure that a remote procedure call application can use.           */
[
uuid(6AF85260-A3A4-101A-B1AE-08002B2E5B76),     /* Universal Unique ID ❶ */
pointer_default(ref)              /* default pointer type is reference ❷ */
]
interface arith                          /* interface name is arith      ❸ */
{
   const unsigned short ARRAY_SIZE = 10; /* unsigned integer constant   ❹ */
```

Example 1-1: A Simple Interface Definition (continued)

```
    typedef long long_array[ARRAY_SIZE];  /* array type of long integers ❺ */

    void sum_arrays (   /* sum_arrays procedure does not return a value ❻ */
        [in] long_array a,                  /* 1st parameter is passed in  */
        [in] long_array b,                  /* 2nd parameter is passed in  */
        [out] long_array c                  /* 3rd parameter is passed out */
    );
}
```

❶ The `uuid` attribute specifies the interface UUID. The interface definition header for any distributed application requires a `uuid` attribute.

❷ RPC provides three types of pointer, offering varying levels of complexity and overhead. Here, the `pointer_default` attribute specifies reference pointers as the default, because they offer the lowest overhead and are sufficient for our purposes.

❸ The last part of the interface definition header contains the keyword `interface` followed by the name chosen for the interface (`arith`).

❹ You can define constants for type definitions and application code. In this example, we define `ARRAY_SIZE` to set the bounds of arrays.

❺ You can define data types for use in other type definitions and procedure declarations. In this example, we define a data type that is an array of ten long integers. The indexes of arrays begin at zero, so the index values for this array range from zero to nine.

❻ The remainder of this interface definition is a procedure declaration. A procedure of type `void` does not return a value. The `in` and `out` parameter attributes are necessary so the MIDL compiler knows in which direction the data need to be sent over the network.

`[in]`: A value is passed in to the remote procedure when it is called from the client.

`[out]`: A value is passed back from the server to the calling procedure on the client when the procedure returns. A parameter with the `out` directional attribute must be a pointer or array so that the parameter can be passed to the client stub by reference. Note that the MIDL compiler requires more complex pointer types to have `[in, out]` attributes.

Stub and Header Generation Using the MIDL Compiler

When the interface definition is complete, you compile it with the MIDL compiler, which creates the following:

- A C language header file that contains definitions needed by the stubs and your application code. You can now include the header file in client and server application code.

- A client stub file, which you will link with the client portion of the application. During a remote procedure call, the client stub code is intermediate between your client application code and the RPC runtime library.
- A server stub file, which you will link with the server portion of the application. During a remote procedure call, the server stub code is intermediate between your server application code and the RPC runtime library.
- Client and server auxiliary stub files linked with the client and server portions of the application. The auxiliary stub files convert complex data structures like pointers to and from a data stream suitable for transmission over the network.

When you invoke the MIDL compiler, it generates the header file and intermediate C language stub files. Although we show a *midl* command by itself here, we recommend that you use a tool like *nmake* and a makefile to automate your entire build procedure. Such tools can hide differences between different hardware platforms making your code more portable. They can also relieve you from the drudgery of typing in long command strings over and over. Later, we'll show a makefile for use in building client and server applications.

To invoke the MIDL compiler and create the header and stub files for the arithmetic interface, type the following:

```
C:\> midl arith.idl
```

In this example, we generate the header file and the C language stub files of the client and server in one operation. The MIDL compiler produces auxiliary stub files by default, but you may suppress their generation by using appropriate MIDL compiler options.

If you develop the client and server on different systems, copies of the interface definition and the MIDL compiler must reside on both the client and server systems. To generate code correctly for different kinds of systems, compile the interface definition for the client stub on the client system, and for the server stub on the server system.

A Simple Client

We'll start our coding with the client, because it's so simple. In fact, you will not be able to detect any difference between our client and a traditional, single-system program! That's one of the beauties about Microsoft RPC—it hides most of the networking complexity from the client developer.

To develop a client, you must be able to read and interpret the interface definition. To use all the capabilities of RPC, you must also know the RPC runtime routines. The client in our simple example, however, requires no RPC runtime routines.

Figure 1-5 shows the files and utilities needed to produce a client. You write the client application code (*client.c*) in C. Currently, Microsoft RPC provides libraries only for C. Remote procedure calls in a client look like local procedure calls. (The

server portion of the application implements the remote procedures themselves.) You must include the header file (*arith.h*) produced by the MIDL compiler, so that its type and constant definitions are available.

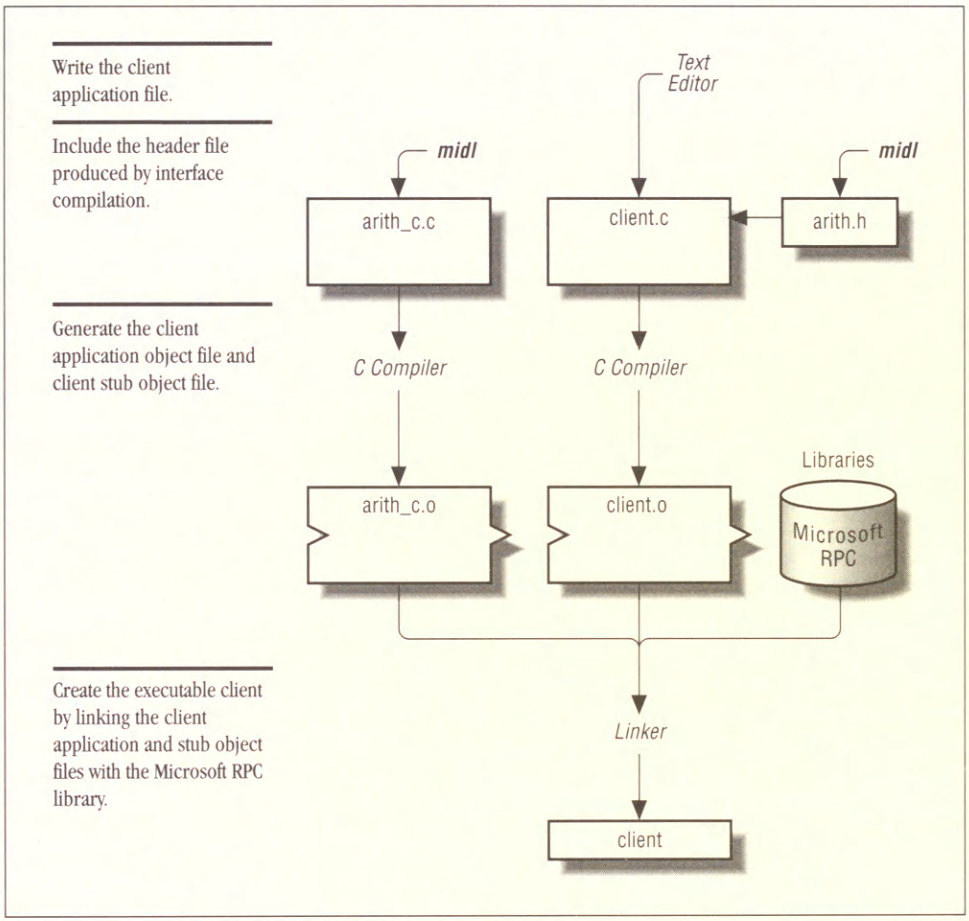


Figure 1-5. Arithmetic application: client development

After compiling *client.c* and *arith_c.c* with the C compiler, you can create the executable client by linking the client stub (*arith_c.o*) with the client object file and the Microsoft RPC library. Example 1-2 shows a simple client.

Example 1-2: A Simple Client

```
/* FILE NAME: client.c */
/* This is the client module of the arithmetic example. */
#include <stdio.h>
#include <stdlib.h>
#include "arith.h"        /* header file created by MIDL compiler ❶ */
```

Example 1-2: A Simple Client (continued)

```
long_array a ={100,200,345,23,67,65,0,0,0,0};
long_array b ={4,0,2,3,1,7,5,9,6,8};

main ()
{
   long_array    result;
   int i;

   sum_arrays(a, b, result);          /* A Remote Procedure Call ❷ */
   puts("sums:");
   for(i = 0; i < ARRAY_SIZE; i++)
      printf("%ld\n", result[i]);
}

/*****************************************************************************/
/***               midl_user_allocate / midl_user_free                     ***/
/*****************************************************************************/

void * __RPC_API
midl_user_allocate              /* Procedures called by the stubs ❸ */
           (
            size
           )
size_t size;
{

    unsigned char * ptr;
    ptr = malloc( size );
    return ( (void *)ptr );

}

void __RPC_API
midl_user_free
           (
            object
           )
void * object;
{
    free (object);
}
```

❶ The client code includes the header file produced by the MIDL compiler.

❷ The client calls the remote procedure *sum_arrays* using the two initialized arrays as input. It then displays the elements of the resulting array.

❸ Two programmer-supplied procedures—*midl_user_allocate* and *midl_user_free*—may be called by client and server stubs for certain memory management functions. Although this simple application does not require these routines, they are essential parts of many Microsoft RPC applications. Usually these are just wrapper routines for *malloc* and *free*. Chapter 4, *Pointers, Arrays, and Memory Usage*, contains more information about these procedures.

The following section shows how to write the server for the arithmetic application.

A Minimal Server

Developing a server requires you to know the interface definition and some RPC runtime routines. You write two distinct portions of code:

- The actual remote procedures—this portion is sometimes called the **manager**
- Code to initialize the server

You make calls to the RPC runtime routines mainly in the server initialization, which prepares the server to listen for remote procedure calls. For our arithmetic application, server initialization is the only code that requires the use of runtime routines.

Figure 1-6 shows the files and utilities needed to produce a server. You must write the remote procedures (*manager.c*) and server initialization code (*server.c*) in C. You need the header file (*arith.h*) produced by the MIDL compiler because it contains definitions required by the remote procedures and runtime calls.

After compiling the server application with the C compiler, you create the executable server by linking the server stub (*arith_s.o*) with the server application object files and the Microsoft RPC library.

Remote Procedure Implementation

The programmer who writes a server must develop all procedures that are declared in the interface definition. Refer to the interface definition (*arith.idl*) and the header file generated by the MIDL compilation (*arith.h*) for the procedure's parameters and data types. Example 1-3 shows the code for the remote procedure of the arithmetic application.

Example 1-3: A Remote Procedure Implementation

```
/* FILE NAME: procedure.c */
/* Implementation of procedure defined in the arithmetic interface. */
#include <stdio.h>
#include "arith.h"          /* header file produced by MIDL compiler ❶ */

void sum_arrays(a, b, c)     /* implementation of sum_arrays procedure ❷ */
   long_array a;
   long_array b;
   long_array c;
   {
   int i;

   for(i = 0; i < ARRAY_SIZE; i++)
      c[i] = a[i] + b[i];    /* array elements are each added together ❸ */
   }
```

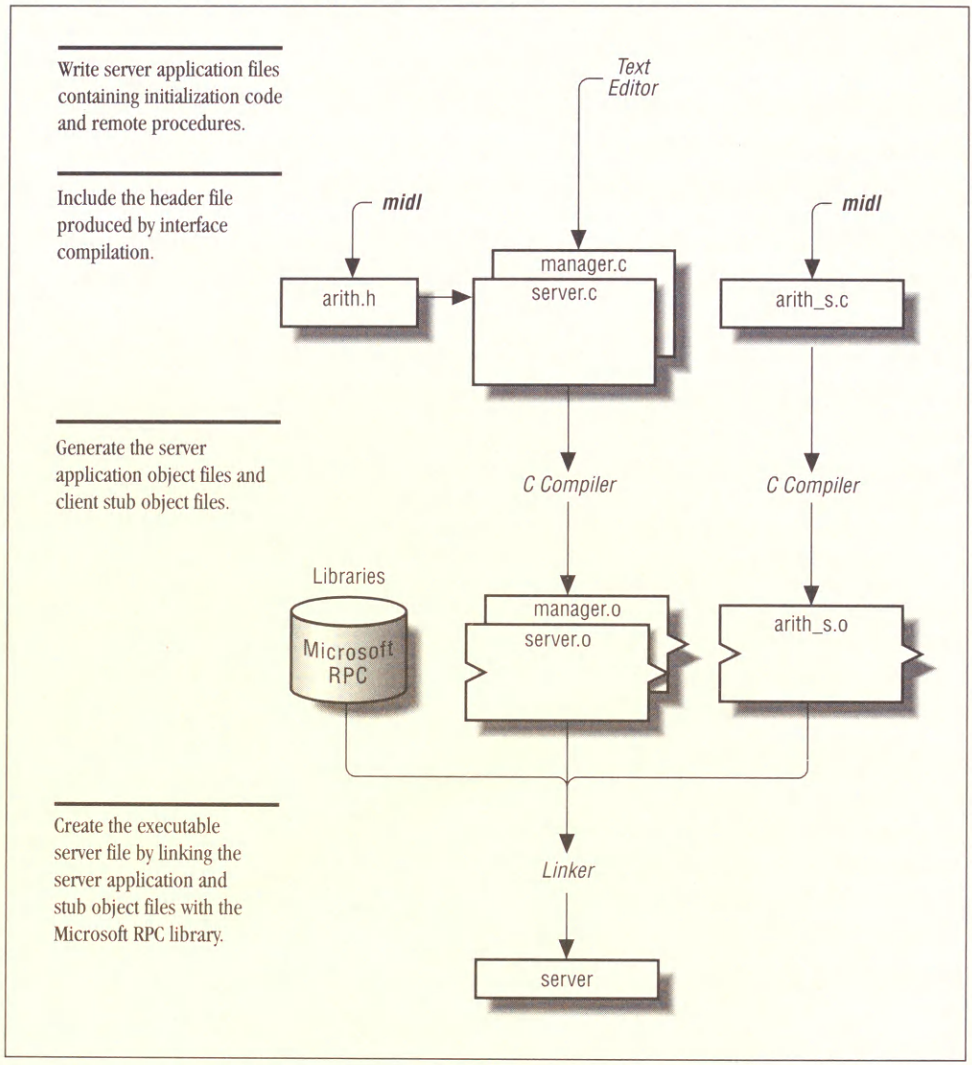


Figure 1-6. Arithmetic application: server development

❶ The server code includes the header file produced by the MIDL compiler.

❷ The procedure definition matches its corresponding declaration in the interface definition.

❸ The procedure implementation is completed.

So far, the client and server application code has been much like any other application. In fact, you can compile and link the client and remote procedures, and run the resulting program as a local test.

Before going on to write the server initialization code, we found it useful to discuss how the arithmetic application works in a distributed environment. This is the subject of the next section.

A Distributed Application Environment

When a client makes a remote procedure call, a binding relationship is established with a server (see Figure 1-7). **Binding information** is network communication and location information for a particular server. Conveniently, in the arithmetic application, the client stub and the RPC runtime library automatically find the server for you during the remote procedure call. Figure 1-8 illustrates how binding information acts like a set of keys to a series of gates in the path a remote procedure call takes toward execution.

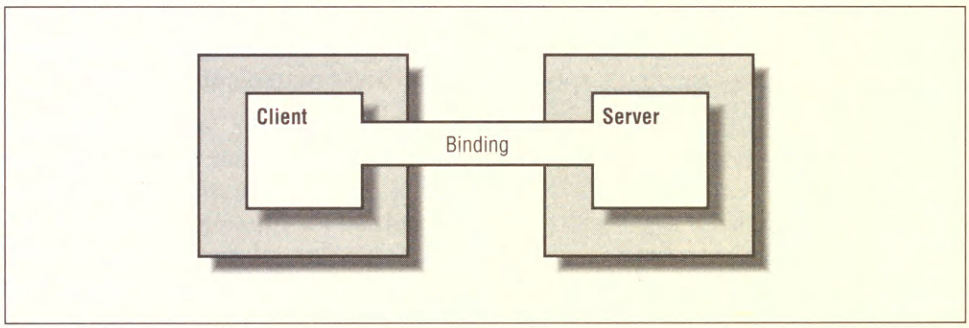


Figure 1–7. Binding

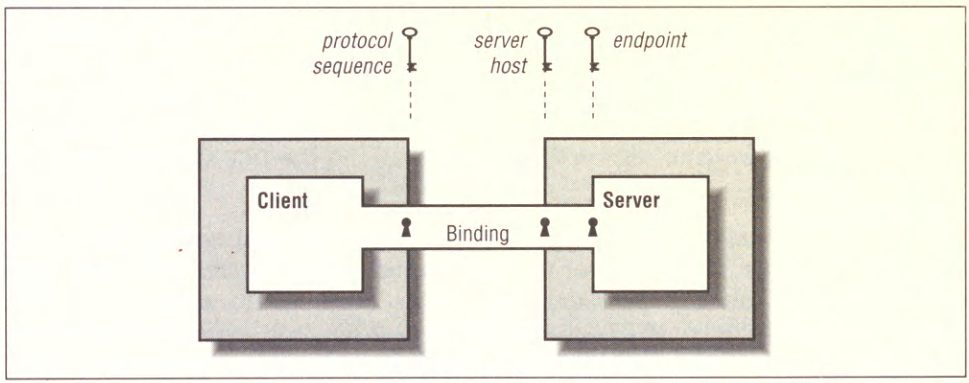


Figure 1–8. Binding information

Binding information includes the following:

1. **Protocol Sequence**

   A protocol sequence is an RPC-specific name containing a combination of communication protocols that describe the network communication used between a client and server. For example, `ncacn_ip_tcp` represents the protocol sequence for a Network Computing Architecture connection-oriented protocol, over a network with the Internet Protocol and the Transmission Control Protocol for transport.

2. **Server Host**

   The client needs to identify the server system. The server host is the name or network address of the host on which the server resides.

3. **Endpoint**

   The client needs to identify a server process on the server host. An endpoint is a number representing a specific server process running on a system.

To help clients find servers in a flexible and portable manner, Microsoft RPC provides a name service to store binding information. **Name service** is a general term for a database service that stores information for distributed applications—that is, a service that offers the same information to applications running on different systems. Using the name service, a server can store binding information that a client on another system can retrieve later. The particular name service offered with Microsoft RPC is called the Locator.

The RPC runtime library contains a general set of functions called **name service independent** (NSI) routines. Thus, to store binding information, your server calls an NSI routine. This routine internally communicates with the Locator to put information into the database. NSI routines are a level of abstraction above the particular name service on a system, and thus can be used to access whatever name service your system uses. For instance, if you shared a network with DCE systems, you could configure your Microsoft RPC system to use the DCE Cell Directory Service (CDS).

Distributed applications do not require the name service database, but we recommend that you use it. Alternatives to using the name service are to manage binding information directly in client and server code, or to create your own application-specific method of advertising and searching for servers. These alternatives present more maintenance problems than if you use the name service routines.

Figures 1-9, 1-10, and 1-11 show how the arithmetic application uses binding information, and how the remote procedure call completes.

A server must make certain information available to clients. Figure 1-9 shows the typical steps needed each time a server starts executing. A server first registers the interface with the RPC runtime library, so that clients later know whether they are

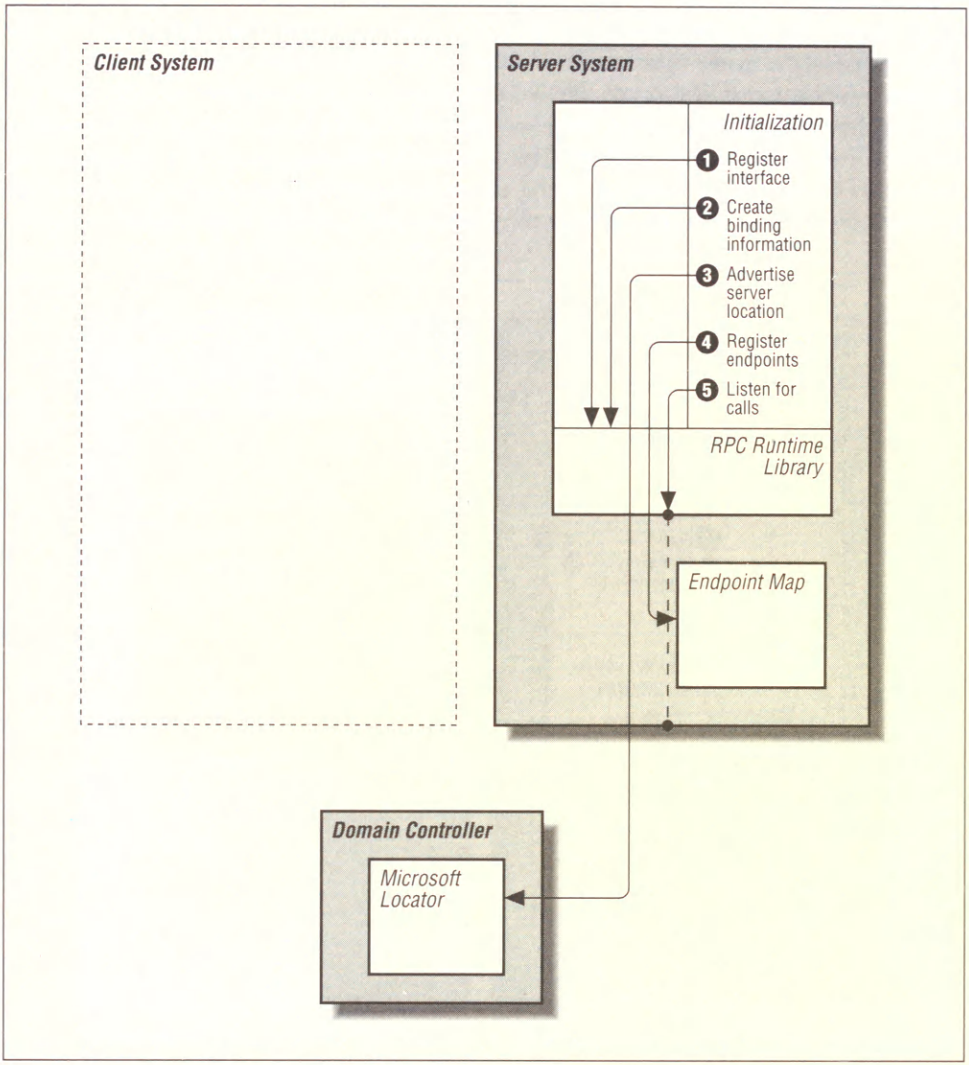


Figure 1–9. Server initializing

compatible with the server. The runtime library creates binding information to identify this server process. The server places the binding information in appropriate databases so that clients can find it. The server places communication and host information in the name service database. The server also places process information (endpoints) in a special database on the server system called the **local endpoint map**, which is a database used to store endpoints for servers running on a given system. In the final initialization step, a server waits while listening for remote procedure calls from clients.

When the server has completed initialization, a client can find it by obtaining its binding information, as illustrated in Figure 1-10. A remote procedure call in the client application code transfers execution to the client stub. The client stub looks up the information in the name service database to find the server system. The RPC runtime library finds the server process endpoint by looking up the information in the server system's endpoint map. The RPC runtime library uses the binding information to complete the binding of the client to the server. Chapter 3, *How to Write Clients*, discusses variations on how to obtain server binding information.

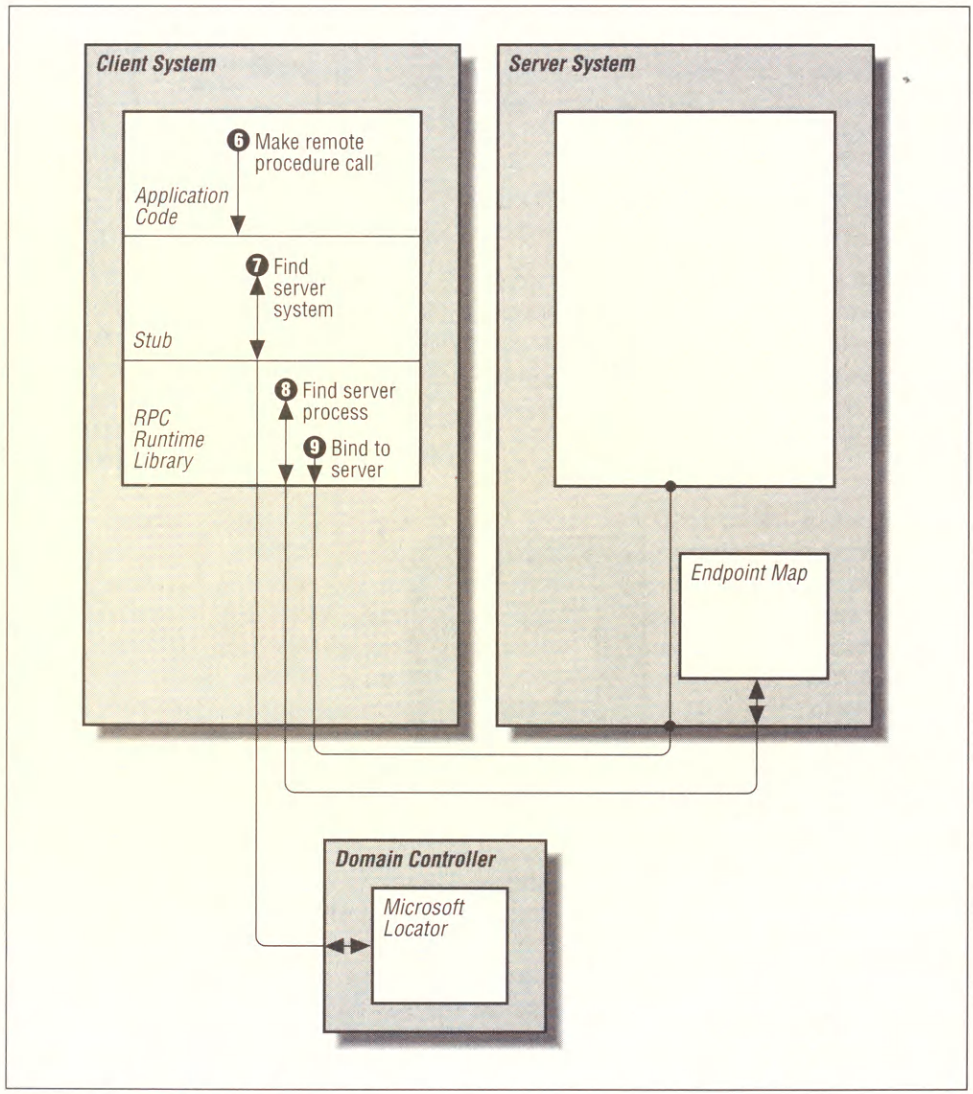


Figure 1-10. Client finding a server

As shown in Figure 1-11, the remote procedure executes after the client finds the server. The client stub puts arguments and other calling information into an internal RPC format that the runtime library transmits over the network. The server runtime library receives the data and transfers it to the stub, which converts it back to a format the application can use. When the remote procedure completes, the conversion process is reversed. The server stub puts the return arguments into the internal RPC format, and the server runtime library transmits the data back to the client over the network. The client runtime library receives the data and gives it to the client stub, which converts the data back for use by the application.

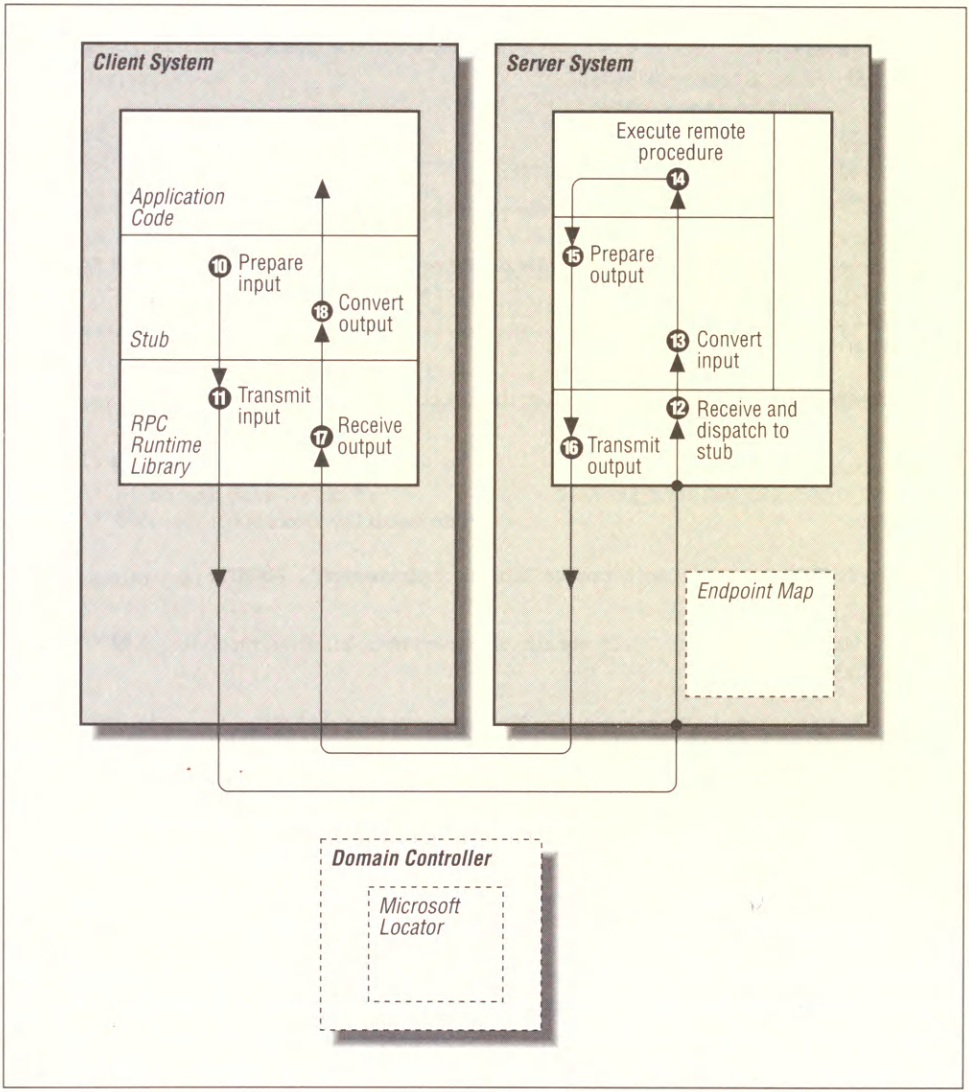


Figure 1-11. Completing a remote procedure call

Server Initialization

As illustrated in Figure 1-9, a server must make certain information available to the RPC runtime library and clients before it can accept remote procedure calls. Example 1-4 contains the server initialization code for the arithmetic application, illustrating the sequence of steps to initialize a typical RPC server.

Example 1-4: A Minimal Server Initialization

```
/* FILE NAME: server.c */
#include <stdio.h>
#include "arith.h"                    /* header created by the MIDL compiler */
#include "status.h"                    /* header with the CHECK_STATUS macro */

main ()
{
   unsigned long        status;                              /* error status */
   rpc_binding_vector_t *binding_vector;           /*set of binding handles */
   unsigned char        *entry_name;          /*entry name for name service */

   status =                                                  /* error status */
   RpcServerRegisterIf(        /* register interface with the RPC runtime ❶ */
      arith_v1_0_s_ifspec,            /* interface specification (arith.h) */
      NULL,
      NULL
   );
   CHECK_STATUS(status, "Can't register interface", ABORT);

   status =
   RpcServerUseAllProtseqs(                /* create binding information ❷ */
      RPC_C_PROTSEQ_MAX_REQS_DEFAULT,                /* queue size for calls */
      NULL                                /* no security descriptor is used */
   );
   CHECK_STATUS(status, "Can't create binding information", ABORT);

   status =
   RpcServerInqBindings(      /* obtain this server's binding information ❸ */
      &binding_vector
   );
   CHECK_STATUS(status, "Can't get binding information", ABORT);

   entry_name = (unsigned char *)getenv("ARITHMETIC_SERVER_ENTRY");
   status =
   RpcNsBindingExport(          /* export entry to name service database ❹ */
      RPC_C_NS_SYNTAX_DEFAULT,      /* syntax of the entry name             */
      entry_name,                   /* entry name for name service          */
      arith_v1_0_s_ifspec,         /* interface specification (arith.h)     */
      binding_vector,               /* the set of server binding handles    */
      NULL
   );
   CHECK_STATUS(status, "Can't export to name service database", ABORT);

   status =
   RpcEpRegister(             /* register endpoints in local endpoint map ❺ */
      arith_v1_0_s_ifspec,        /* interface specification (arith.h)      */
```

Example 1-4: A Minimal Server Initialization (continued)

```
        binding_vector,              /* the set of server binding handles      */
        NULL,
        NULL
    );
    CHECK_STATUS(status, "Can't add address to the endpoint map", ABORT);

    status =
    RpcBindingVectorFree(            /* free set of server binding handles ❻ */
        &binding_vector
    );
    CHECK_STATUS(status, "Can't free binding handles and vector", ABORT);

    puts("Listening for remote procedure calls...");
    status =
    RpcServerListen(                          /* listen for remote calls ❼ */
        1,                                    /* minimum number of threads  */
        RPC_C_LISTEN_MAX_CALLS_DEFAULT,       /* concurrent calls to server */
        NULL                    /* continue listening until explicitly stopped */
    );
    CHECK_STATUS(status, "rpc listen failed", ABORT);
}

/*****************************************************************************/
/***              midl_user_allocate / midl_user_free                      ***/
/*****************************************************************************/

void * __RPC_API
midl_user_allocate            /* Procedures called by the stubs          ❽ */
            (
             size
            )
size_t size;
{

    unsigned char * ptr;
    ptr = malloc( size );
    return ( (void *)ptr );

}

void __RPC_API
midl_user_free
            (
             object
            )
void * object;
{
    free (object);
}
```

❶ **Register the interface.** Register the interface with the RPC runtime library using the *RpcServerRegisterIf* routine. The `arith_v1_0_s_ifspec` variable is called an **interface handle**. It is produced by the MIDL compiler and refers to information that applications need, such as the UUID. We describe the NULL arguments in Chapter 5, *How to Write a Server*.

The CHECK_STATUS macro is defined in the *status.h* header file for the applications in this book. It is used to interpret status codes from runtime calls. (See Example 3-12 in Chapter 3.) Figure 1-9, step 1 is now complete.

❷ **Create binding information.** To create binding information, you must choose one or more network protocol sequences. This application, like most, calls *RpcServerUseAllProtseqs* so that clients can use all available protocols. During this call, the RPC runtime library gathers together information about available protocols, your host, and endpoints to create binding information. The system allocates a buffer for each endpoint, to hold incoming call information. Microsoft RPC sets the buffer size when you use the *RPC_C_PROTSEQ_MAX_CALLS_DEFAULT* argument.

❸ **Obtain the binding information.** When creating binding information, the RPC runtime library stores binding information for each protocol sequence. A **binding handle** is a reference in application code to the information for one possible binding. A set of server binding handles is called a **binding vector**. You must obtain this information through the *RpcServerInqBindings* routine in order to pass the information to other runtime routines. Figure 1-9, step 2 is now complete.

❹ **Advertise the server location in the name service database.** In this example, the server places (exports) all its binding information in the name service database using the *RpcNsBindingExport* runtime routine.

The *RPC_C_NS_SYNTAX_DEFAULT* argument tells the routine how to interpret an entry name. (The current version of Microsoft RPC has only one syntax.) The `entry_name` is a string obtained in this example from an environment variable set by the user specifically for this application, ARITHMETIC_SERVER_ENTRY (discussed at the end of this chapter when the application is run). The interface handle, `arith_ServerIfHANDLE`, associates interface information with the entry name in the name service database. The client later uses name service routines to obtain binding information by comparing the interface information in the name service database with information about its own interface. Figure 1-9, step 3 is now complete.

❺ **Register the endpoints in the local endpoint map.** The RPC runtime library assigns endpoints to the server as part of creating binding information. The *RpcEpRegister* runtime routine lets the endpoint map on the local host know that the process running at these endpoints is associated with this interface. Figure 1-9, step 4 is now complete.

❻ **Free the set of binding handles.** Memory for the binding handles was allocated with a call to the *RpcServerInqBindings* routine. When you have finished passing binding information to the other routines, release the memory using the *RpcBindingVectorFree* routine.

❼ **Listen for remote calls.** Finally, the server must wait for calls to arrive. Each system has a default for the maximum number of calls that a server can accept at one time. Microsoft RPC sets this maximum default number when you use the *RPC_C_LISTEN_MAX_CALLS_DEFAULT* argument. Figure 1-9, step 5 is now complete.

❽ Two programmer-supplied procedures—*midl_user_allocate* and *midl_user_free*—may be called by client and server stubs for certain memory management functions. Although this simple application does not require these routines, they are essential parts of many Microsoft RPC applications. Usually these are just wrapper routines for *malloc* and *free*. Chapter 4 contains more information about these procedures.

All of the server code is now complete. The compilation of the application is shown in the next section.

Producing the Application

So far we have written the interface definition, produced the stubs and header file from the interface definition with the MIDL compiler, and written the client and server portions of the application. To produce the application, compile and link the client and server separately, each on the system where you want its executable to run.

Microsoft RPC Libraries

Microsoft RPC-distributed applications must be linked with the Microsoft RPC libraries, which may vary depending on your system and vendor. This book uses the following libraries for a link on a Microsoft Windows NT system:

```
rpcrt4.lib
rpcns4.lib
libcmt.lib
kernel32.lib
```

The *rpcrt4.lib* library provides Windows runtime library functions. The *rpcns4.lib* library provides name service functions. The *libcmt.lib* library provides standard C library functions. The *kernel32.lib* library provides threads functions.

The following sections assume that your client and server files are available to the respective client and server systems.

Compile and Link the Client and Server Code

Recall that Figures 1-5 and 1-6 show the utilities used and files produced when developing a client and a server. Here, we show a portion of a makefile we use